

REMARKS

Applicant wishes to thank the Examiner for the attention accorded to the instant application, and respectfully requests reconsideration of the application based on the following remarks.

Formal Matters

Claims 76-98 are the claims currently pending in the Application.

Response To Rejections Under 35 U.S.C. § 102

The Examiner rejected claims 76-98 under 35 U.S.C. § 102(e) as being anticipated by Krishnan et al., U.S. Patent 6,141,698 (hereinafter "Krishnan"). This rejection should be withdrawn based on the comments and remarks herein.

Among the problems recognized and solved by Applicant's claimed invention is the need for an automated technique for enhancing an executable file to extend its functionality. The present invention solves this need by providing a method for inserting new code between arbitrary instructions of the original executable code.

Despite employing a somewhat similar nomenclature, e.g., the term "injection" for adding new functionality to an existing piece of software, the present invention is patentably distinguishable from Krishnan for the reasons discussed below.

Krishnan discloses an anti-piracy scheme for executables that is threefold. First, the disclosed scheme adds *protection code* to the original executable code. Second, the disclosed scheme increases the robustness of the protection code against manipulation and removal by adding *security code*. Third, the disclosed scheme further increases the robustness of the protection code against manipulation and removal by *incremental encryption and decryption* of portions of the original executable code.

1. Adding protection or licensing code

Krishnan teaches that protection or licensing code to be injected is kept in a DLL, i.e. in a file that is external to the file that contains the original executable code (column 3, lines 60-63, Figure 4). Krishnan offers two methods for invoking the functionality contained inside the external DLL, the *import table technique* and the *DLL loader code technique* (column 4, lines 17-31), which are compared to the present patent application in the following two sections. Both techniques allow a block of code to be executed before execution of the original executable code. So, the separation of the protection code from the original executable code is complete not only with respect to its storage location, i.e. executable file vs. external DLL file, but also with respect to execution order, i.e. the protection code is always executed before the original application code.

The import table technique, shown in Figure 3, modifies metadata, or the import table, of the executable file to be processed to reference the external DLL that contains the code to be executed before the original executable code inside the executable file. This method leaves the original executable code contained in the executable file completely untouched. After processing, the executable file contains exactly the same code as before processing. Figure 3 clearly illustrates that only the import table is modified. As it is not relevant, the original executable code is not even identified in the figure.

The DLL loader code technique taught by Krishnan (see column 8, line 24 to column 9 line 6) modifies the code entry point of the executable file to chain the execution of the protection code inside the external DLL and the original executable code inside the executable file. The code entry point specifies where inside the executable file

the operating system is supposed to start execution when the executable file is run by a user. For this, the code entry point identifies the first instruction of the executable file to be executed.

The DLL loader code technique adds a block of code, the *DLL loader code*, to the executable file. The code entry point of the executable file is then modified to point to the first instruction of the newly added block of code instead of the block of code that constitutes the original executable code. After the newly added block of code has been executed, it triggers execution of the block of code that constitutes the original executable code.

While code is added to the executable file, the individual instructions of the original executable code are not considered and are left untouched. The original executable code as well as the newly added DLL loader code constitute completely separate monolithic blocks. As such, they are located in two different locations, i.e. their instructions are not interleaved, and the DLL loader code is always executed before the original executable code. No instruction-level analysis is performed.

Figure 6 of Krishnan illustrates the DLL loader technique. Originally the code entry point references instruction 1 of the instructions that constitute the original application code (instruction 1, instruction 2, ... instruction n). After the transformation, the code entry point references the first instruction of the added DLL loader code, which in turn references instruction 1 of the original application code. Note that the original executable code remains unchanged, consisting of instruction 1 through instruction n before and after the transformation; the original application code is merely considered as

a block. Hence, Krishnan's disclosed scheme does not operate on individual instructions when it adds new code to existing code.

2. Security Code

The security code technique taught by Krishnan, shown in Figure 9, performs integrity checks on the executable file as well as the external DLL file. Just as in the DLL loader code technique, the executable file is handled as a single block instead of individual instructions.

If the import table technique is used to invoke the code supplied by the external DLL file, then the security code is added to the executable file just like the DLL loader code would have been added had the DLL loader code technique been employed, hence the similarity between figures 8A and 6. In both cases, the code entry point is modified to reference the newly added code (8A05 and 607, respectively), and the newly added code then references the original executable code (8A08 and 606, respectively). Adding the DLL loader code does not differ from adding the security code and, thus, the above analysis of DLL loader code insertion also applies to security code insertion. No instruction-level analysis is performed; the original executable code as well as the newly added code is processed as an opaque block, and not on the instruction level.

If the DLL loader code method is used to invoke the code supplied by the external DLL file, then the security code is added to the executable file in the same way as the DLL loader code. However, the security code is inserted into the executable file in addition to the DLL loader code and not in lieu of the DLL loader code. Still, the same method is used for inserting the DLL loader code and the security code. Therefore, again in this case, the above analysis of DLL loader code insertion also applies to security code

insertion. No instruction-level processing occurs. Hence, for security code insertion, the security code as well as the original executable code is handled as a whole, i.e., as an opaque block, and not as individual instructions.

3. Incremental encryption and decryption

Incremental encryption and decryption, as taught by Krishnan, divide the original executable code into encrypted blocks or a set of “conceptual chunks” of instructions (column 12, lines 6-15). This is the only situation in which Krishnan teaches considering the individual instructions that constitute a block of executable code instead of merely working on the entire block of executable code as a whole. However, the analysis of the instructions, as taught by Krishnan, is performed solely to optimally divide the original executable code into said set of conceptual chunks (column 13, lines 18-21). The idea is that in each chunk, there is a single instruction through which the control flow enters the chunk and a single instruction through which the control flow leaves the chunk (column 13, lines 42-45). The corresponding algorithm, illustrated in Figure 12, starts with a default chunk size and enlarges the chunk until the above constraints hold for the chunk. In contrast to applicant’s application, however, Krishnan does not use the results of this analysis to add new code to the original executable code.

While Krishnan teaches a concept of modifying the original executable code at runtime during incremental decryption, as illustrated by figure 14, Krishnan is distinguishable from the present application. Krishnan discloses that the memory locations immediately following the chunk of code to be decrypted are temporarily overwritten with an instruction that redirects the control flow to a memory location marked as “LABEL” (column 16, lines 33-44). Significantly, this modification happens

at runtime and not at processing time. Moreover, it does not *insert* a new instruction, which would imply moving or relocating all subsequent instructions to create space for the new instruction to be inserted into. Instead, the presented modification simply *overwrites* the memory space immediately following the current chunk.

To summarize, Krishnan discloses several methods for modifying the code of an executable file by adding new code. These methods handle the original executable code as well as the new code to be added as a whole, i.e. as monolithic blocks, and they do not consider the fact that the processed blocks consist of instructions. Krishnan also discloses a method for incremental encryption and decryption of the original executable code. This method does consider the fact that the original code consists of instructions. However, the insight gained from the instruction-level analysis of the original executable code is not used for adding code. Instead, it is solely used to determine boundaries for grouping the original executable code into conceptual chunks for the purpose of encryption.

In contrast to these known methods, the present invention allows for inserting new code between arbitrary instructions of the original executable code. This is neither known from nor suggested by Krishnan. In particular, Krishnan does not disclose or suggest "relocating all instructions and all variables within said executable file affected by the insertion, and adjusting all references to the relocated instructions and/or variables to reflect the relocating of the affected instructions and variables" as recited in independent claims 76 and 98.

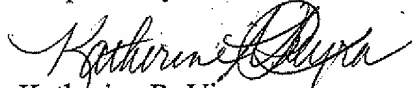
It has been held by the courts that "Anticipation requires the presence in a single prior art reference disclosure of each and every element of the claimed invention,

arranged as in the claim.” *Lindemann Maschinenfabrik GMBH v. American Hoist and Derrick Company et al.*, 730 F.2d 1452, 221 USPQ 481 (Fed. Cir. 1984). As illustrated above, Krishnan does not disclose relocating all instructions and all variables within said executable file affected by the insertion, and adjusting all references to the relocated instructions and/or variables to reflect the relocating of the affected instructions and variables, and does not disclose each and every feature of the invention as recited in independent claims 76 and 98. Claims 77-97 depend from independent claim 76, thus incorporating novel and nonobvious features of the base claims. Accordingly, claims 77-97 are patentably distinguishable over the art of record in the application for at least the reasons that independent claim 76 is patentably distinguishable over the art of record in the application. Therefore, this rejection should now be withdrawn.

Conclusion

For at least the reasons set forth in the foregoing discussion, Applicant believes that the Application is now allowable, and respectfully requests that the Examiner reconsider the rejection and allow the Application. Should the Examiner have any questions regarding this Response, or regarding the Application generally, the Examiner is invited to telephone the undersigned attorney.

Respectfully submitted,



Katherine R. Vieyra
Registration No. 47,155

SCULLY, SCOTT, MURPHY & PRESSER, P.C.
400 Garden City Plaza, Suite 300
Garden City, New York 11530
(516) 742-4343

KRV/jam